Spring WebFlow

- Web application page flow can become complex very easily, requiring you to have inside knowledge of an application's implementation just to understand the flow through the application.

- It is virtually guaranteed that anyone developing web applications will experience this problem at some point.

- By and large, the most common way to deal with page flow is to hard-code it into the application.
  - However, hard-coding page flow is not a good idea for many reasons.
  - The most important one is that understanding such a page flow requires a detailed comprehension of a web application's inner workings.
  - Depending on the size of the web application and the complexity of its page flow, acquiring detailed knowledge of the architecture can be difficult and time-consuming.

- But what if you were able to capture all page flow details in a single location?

- The business rules driving the design of applications can run the range from very simple to highly complex. But generally there are three types of web applications:
  1. *Web applications that start out with a simple set of goals and remain simple*
  2. *Web applications that start out with a simple set of goals and grow in complexity over time*
  3. *Web applications that start out with a complex set of goals*

1. Web applications that start out with a simple set of goals and remain simple:

- *These applications* are made to fulfill a small but meaningful need and never grow beyond that.

- Simple and to the point, these applications have no need of additional features.

2. Web applications that start out with a simple set of goals and grow in complexity over time

- *These* applications aren't originally designed to conquer a massive set of complex goals.

- They typically have simple beginnings, maybe intended to solve a small part of a much larger, more complex problem.

- Examples of these applications run far and wide in the corporate world — for example, a small piece of a supply chain application, a minor addition to a customer relationship management (CRM) application, or even a simple integration of two systems.

- Over time, these applications can take on a life of their own and grow far beyond the original intentions of the designers.

- These applications typically are developed inside a business or an organization.

3. Web applications that start out with a complex set of goals

- *These applications are usually tackled* by an entire team of people and can have a fairly long implementation time.

- Examples of such applications might include a reservation system, a loan origination system, or even a system to categorize media.

- These systems are inherently complex in their design because they are designed to model each process in a way that allows humans to easily understand and interact with each one.

- Of course, the goal of making a process easier to understand is not always achieved, because simplifying a complex process is itself rather difficult.

□ Consider these three styles of web applications and then think about the page flow for each one.

□ Any need to define a formal page flow?

☐ Style one probably has no need to define a formal page flow;

☐ styles two and three definitely do.

☐ But who's to say which applications need a formal page flow and which do not?

   ◻ There's no single test to determine this, so it's like anything else — a judgement call.

   ◻ The best way to decide is to understand the problems solved by a formal page flow mechanism to see if it will benefit your projects.

Examining a Sample Work Flow for Loan Applications

- Examples of flows abound throughout the business world. Just about any business process can be identified and better examined through the use of a proper process flow diagram.

- Consider even simple flows in the business world, such as those involved in ordering office supplies, paying vendors, or filing taxes.

- Even though these tasks may not seem to need a process flow diagram, each one still has its own set of states, transitions and actions.

- In the interest of saving money, efficiency experts could diagram them to intricately understand each step involved. But process flow diagrams aren't just limited to helping you understand a process' efficiency.

- In order to understand flows better, let's look at some examples a bit closer now

- Next slide shows a simplified, high-level flow for a loan application system.
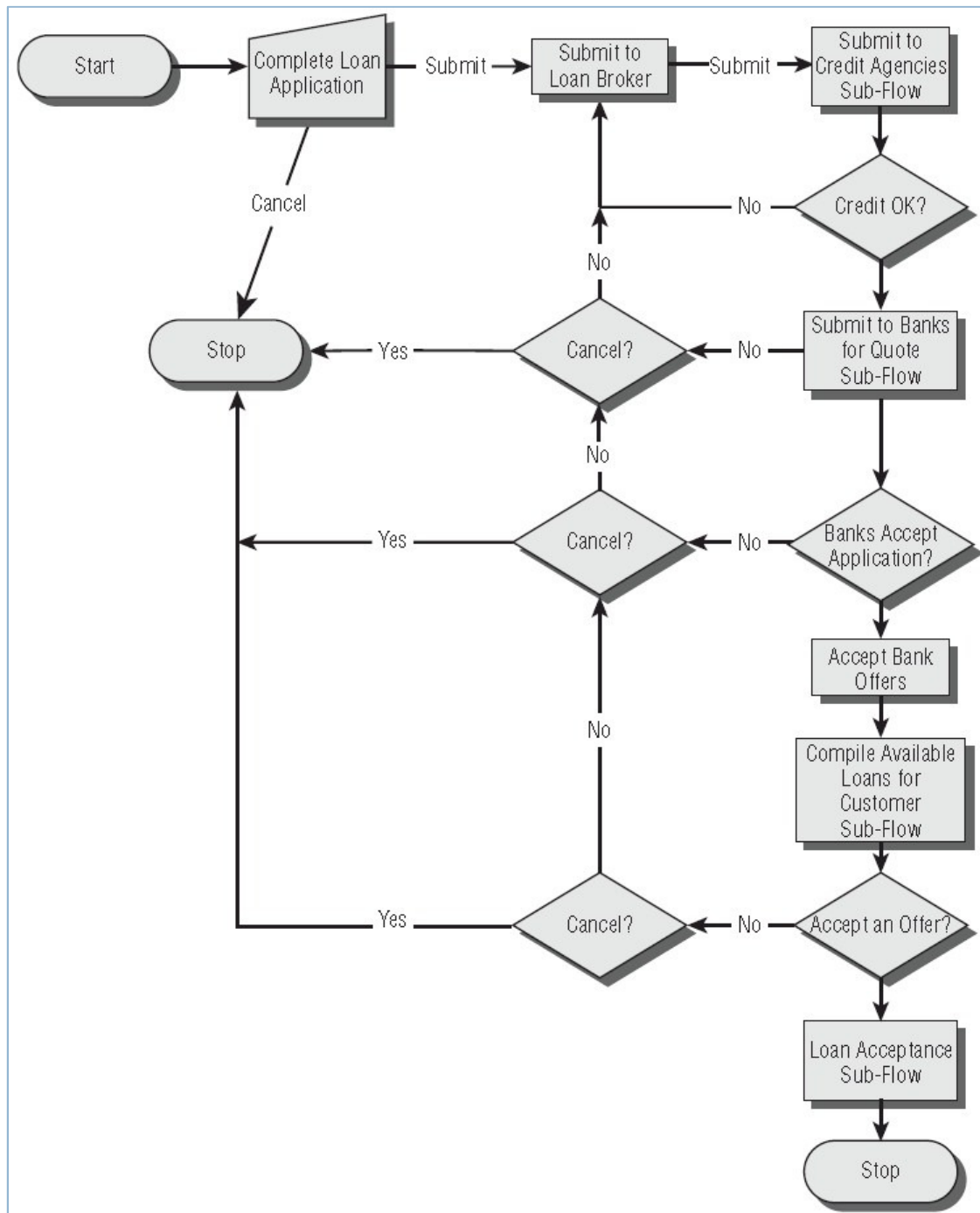
Loan Application

- This system is complex enough to be suitable for using Spring Web Flow (SWF).
    - This flow has many states, transitions, and actions.
    - Formally defining these different attributes enables you to identify subflows and reuse them where necessary.
- This makes the development of the application that much faster and means that new developers to the project will be able to understand the flow much more quickly than if the flow had been hard-coded deep in the internals of the application.
- Additional examples of flows include voting systems, supply chain management systems, assembly lines, manufacturing systems, and even media categorization system
- All of these systems are complex by nature and couldn't easily be addressed through the use of a MVC framework alone — that would require an intimate understanding of how the flow is implemented using the MVC framework, which in turn would mean tightly coupling the MVC framework to the application logic. That is not a good decision
- Describing processes independently of the MVC framework allows any web application or rich client application to use the formally defined flow as the workflow of an application.
- One such formal workflow mechanism is Spring Web Flow (SWF).

Introducing Spring Web Flow

- SWF allows for the definition of self-contained controller configurations, known as *flows, that define a* conversational dialog between user requests and application responses.
- *Flow is a term used to identify the* movement in a web application from one page to another.
- SWF is designed to handle states, transitions, and events.
- SWF does this through the implementation of a specialized controller specifically for flows, named *FlowController.*
- FlowController is used to execute all flows. It serves as an engine for tracking states, moving among those states, invoking actions, and handling events.
- Flows can be defined using XML or programmatically using Java. The separation provided by defining a flow via XML externalizes the flow, separates it from the controller, and allows for its reuse in other flows.
- A flow definition consists of various steps, denoted by states in the flow transitioning from one state to another based on events.
- SWF begins when a user launches an application.
- This in turn causes a flow to be executed and SWF enters the start state of that flow.
- From there, the flow definition determines what happens next.

How SWF Works with Spring MVC

- It is very logical to wonder how SWF works with Spring MVC because of the overlap each has with the other in certain areas.
- The perception of an overlap is true, it does exist, but Spring Web Flow is designed to *complement Spring MVC so that your web applications can define reusable flows that can be employed* in areas beyond just the HttpServletRequest and HttpServletResponse.
- First let's take a look at the most obvious overlap: multipage flow.
- Spring MVC provides two form controllers;
  - SimpleFormController and
  - AbstractWizardFormController.
- The AbstractWizardFormController simplifies building applications that use multipage forms and is typically used for the processing of *sequential page flow — and this is a major difference.*
- *The polarity between a sequential page flow and a* freeform conversational page flow are subtle but dramatically different.
- When a page flow can progress only from page one to page two and then to page three, and so on, the AbstractWizardFormController is completely appropriate.
- When a page flow contains many states and transitions similar to those shown in the Loan Sample Application, it is much more appropriate to use a formal flow definition using SWF.

□ Spring Web Flow is an extension to Spring MVC (actually, it's just another controller) that provides for the development of conversation-style navigation in a web application.

The key features provided for by Spring Web Flow are:
- ■ The ability to define an application's flow external to the application's logic
- ■ The ability to create reusable flows that can be used across multiple applications

Flow definition is the first step to using SWF

❑ *Flows: A flow defines a* conversation between the user and the application. *Flows are the base* artifact of SWF. They are used to configure SWF and determine its behavior. A flow consists of a set of states. The loan application flow is an example of such a flow.

❑ *States: The steps in a flow are known as states. A state is a point in the flow at which some sort* of action occurs. Each state has one or more transitions used to proceed to the next state. There are several state types in Spring Web Flow:

  ❑ *Start state: Each flow has exactly one start state.*

  ❑ *Action state: A state for the execution of application logic.*

  ❑ *View state: Used to display a view to the user of the application.*

  ❑ *Decision state: Uses a condition to determine the transition to another state.*

  ❑ *Subflow state: A state for the execution of another flow.*

  ❑ *End state: Represents the termination of a flow.*

❑ *Transitions: Transitions are changes from state to state and are initiated by events.*

❑ *Events: An event is the occurrence of something important. Typically an event represents the outcome of executing a state.*

These concepts are central to understanding SWF because they are the base of SWF architecture. Beyond these base concepts, SWF is also very focused on views within an application and therefore makes heavy use of Spring MVC. So what is SWF's relation to Spring MVC?

Spring Web Flow's selection of states.

State type	XML element	What it's for
Action	`<action-state>`	Action states are where the logic of the flow takes place. Action states typically store, retrieve, derive, or otherwise process information to be displayed or processed by subsequent states.
Decision	`<decision-state>`	Decision states branch the flow in two or more directions. They examine information within flow scope to make flow routing decisions.
End	`<end-state>`	The end state is the last stop for a flow. Once a flow has reached end state, the flow is terminated and the user is sent to a final view.
Start	`<start-state>`	The start state is the entry point for a flow. A start state does not do anything itself and effectively serves as a reference point to bootstrap a flow.
Subflow	`<subflow-state>`	A subflow state starts a new flow within the context of a flow that is already underway. This makes it possible to create flow components that can be composed together to make more complex flows.
View	`<view-state>`	A view state pauses the flow and invites the user to participate in the flow. View states are used to communicate information to the user or to prompt them to enter data.

Spring WebFlow States

- The selection of states provided by Spring Web Flow makes it possible to construct virtually any arrangement of functionality into a conversational web application.
- While not all flows will require all of the states described in the previous table, you'll probably end up using most of them at one time or another.
- Of all the states in the table, you'll probably find that your flow is mostly composed of view states and action states.
- These two states represent each side of a conversation between the user and the application.
- View states are the user's side of the conversation, presenting information to the user and awaiting their response.
- Action states are the application's side of the conversation, where the application performs some application logic in response to a user's input or the results of another action state.
- Once a state has completed, it fires an *event.*
- *The event is simply a String value* that indicates the outcome of the state.
- By itself, the event fired by a state has no meaning.
- For it to serve any purpose, it must be mapped to a *transition.*
- *Whereas a* state defines an activity within a flow, transitions define the flow itself.
- A transition indicates which state the flow should go to next.

Spring MVC Shortcomings

- To understand Spring MVC's shortcomings with regard to conversational applications, let's suppose that the phone call between the pizzeria employee and the customer were to take place in an online pizza order entry application.

- If we were to build the pizza order entry application using Spring MVC, we'd likely end up coding the application's flow into each controller and JSP.

- For example, we might end up with a JSP with a link that looks like this:
  `<a href="addPizza.htm">Add Pizza</a>`

- As for the controller behind that link, it may be a form controller like this one:

```
public class AddPizzaController extends SimpleFormController
{
    public AddPizzaController() { }
    protected ModelAndView onSubmit(Object command, BindException bindException) throws
        Exception
    {
        Pizza pizza = (Pizza) command;
        addPizzaToOrder(pizza);
        return new ModelAndView("orderDetail");
    }
}
```

- While there's nothing inherently wrong with that link or the controller, it isn't ideal when taken in the context of the conversational pizza application.

- That's because both the link and the controller know too much.

- The link's URL and the controller's ModelAndView each hold a piece of information about the appli cation's flow.

- But there's no place to go to see the complete picture; instead, the application's flow is embedded and scattered across the application's JSPs and controllers.

- Because the flow definition is sprinkled throughout the application's code, it's not easy to understand the overall flow of the application.

- To do so would require viewing the code for several controllers and JSPs.

- Moreover, changing the flow would be difficult because it would require changing multiple source files.

What about AbstractWizardFormController?

- At this point, you may be wondering why Spring Web Flow is necessary when we have AbstractWizardFormController.

- At first glance, AbstractWizardFormController seems to be a way to build conversation into a Spring MVC application.

- But upon closer inspection you'll realize that AbstractWizardFormController is just a form controller where the form is spread across multiple pages.

- Moreover, the "flow" of a subclass of AbstractWizardFormController is still embedded directly within the controller implementation, making it hard to discern flow logic from application logic.

- Spring Web Flow loosens the coupling between an application's code and its page flow by enabling you to define the flow in a separate, selfcontained flow definition.

Launching Flows

- The FlowController handles all flow execution for SWF and is configured as a normal JavaBean, as any Spring Web MVC controller would be.
- Let's walk through the configuration of SWF as it applies to a web application that uses Spring MVC.
- This configuration is located in a file named pix-webflow-config.xml, which is read by the Spring DispatcherServlet from the web.xml file upon startup of the web application and is used to configure the FlowController bean.

```
<!-- Creates the registry of flow definitions for this application -->
<flow:registry id="flowRegistry">
<flow:location path="/WEB-INF/flows/**-flow.xml" />
</flow:registry>


<!--Launches new flow executions and resumes existing executions -->
<flow:executor id="flowExecutor" registry-ref="flowRegistry" />


<!-- Map all requests to /flow.htm to the flow controller to work against the flow registry -->
<bean name="/flow.htm" class="org.springframework.webflow.executor.mvc.FlowController">
    <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

FlowController

- The FlowController is an extension to the Spring AbstractController that serves as a point of integration between Spring MVC and SWF.
- Its job is to handle the execution of flows that are stored in the FlowRegistry.

- The flow registry serves as an index of flows.
- In the previous case, the flow registry is reading in all the flows in the /WEB-INF/flows directory that match the pattern **-flow.xml.

- Views are still resolved by Spring MVC and the registered flows are executed via events.

- Using events is very powerful and extremely flexible because you define all your own events.

- The web flow controller examines the request parameters to figure out what action to take.
- The three recognized parameters are:
  - flowId: The ID of a flow for which a new execution should begin
  - flowExecutionKey: The ID of an ongoing flow execution
  - eventId: The event to be triggered in the current state of the flow; required for ongoing flow execution

- Flows are identified in views by means of the _flowExecutionId parameter and flow events are triggered by means of the _eventId parameter.

An example of a view form containing these parameters:

...

```
<body>
<form:form>
...
<input type="hidden" name="_flowExecutionId" value="${flowExecutionKey}" />
<input type="button" name="_eventId_submit" value="Login" />
</form:form>
</body>
...
```

- The ${flowExecutionKey} is dereferenced into a unique key representing a particular flow located in the flow registry.

- The _eventId_ submit parameter is used by Spring MVC and the submit portion is used as the event being triggered.

- How that event is handled is up to the transition that handles it.

Scopes supported by SWF

□ Spring MVC is specifically designed for handling the control and data validation with the Servlet API, whereas SWF is specifically designed for defining and handling page flows.

□ The Servlet API provides access to request and session scope, whereas SWF provides access to its own scope.

□ Following are the scopes supported by SWF:
❑ *Request:*
> *A scope that is local to a single request in a web flow*
> *once that request completes,* the scope is destroyed.

❑ *Flash: A scope that is available until the next user event that survives refreshes is signaled*

❑ *Flow: This scope exists for the life of a flow session.*

❑ *Conversation: A scope that is available until the root session for flow ends*

Laying the Flow Groundwork

□ we're going to build an application for pizza order entry using Spring Web Flow.

□ In doing so, you'll see that we'll be able to define the application's flow completely external to the application code and views.

□ This will make it possible to rearrange the flow without requiring any changes to the application itself.

□ It will also aid in understanding the overall flow of the application because the flow's definition is contained in a single location.

□ To define the pizza order flow, we'll start by creating a skeleton flow definition file. It'll start out rather empty, but it will be full of state and transition definitions.

□ Flows in Spring Web Flow are defined in XML. Regardless of the specifics of the flow, all flow definition files are rooted with the <flow> element:

<flow>

. . .

</flow>

□ The first thing you should understand about Spring Web Flow definition files is that they are based on a completely different XML schema than the Spring container configuration file.

□ Where the Spring configuration file is used to declare <bean>s and how they are related to each other, a Spring Web Flow definition declares flow states and transitions.

Flow Variables

- The whole purpose of the pizza order flow is to build a pizza order.
- Therefore, we'll need a place to store the order information.
- Below is the Order class, a simple domain bean for carrying pizza order information.

```java
public class Order implements Serializable
{
    private Customer customer;
    private List<Pizza> pizzas;
    private Payment payment;

    public Order() {
        pizzas = new ArrayList<Pizza>();
        customer = new Customer();
    }
    public Customer getCustomer() {
        return customer;
    }
    public void setCustomer(Customer customer) {
        this.customer = customer;
    }
    public List<Pizza> getPizzas() {
        return pizzas;
    }
    public void setPizzas(List<Pizza> pizzas) {
        this.pizzas = pizzas;
    }
    public void addPizza(Pizza pizza) {
        pizzas.add(pizza);
    }
    . . . .
    //other getters and setters
}
```

- So that the Order object is available to all of the states in the flow, we'll need to declare it in the flow definition file with a <var> element:

  <var name="order" class="com.springinaction.pizza.domain.Order" scope="flow"/>

- Notice that the scope attribute is set to flow.

- Flow variables can be declared to live in one of four different scopes

  (*Request, Flash, Flow, Conversation*)

- Since we'll need the Order object throughout the entire life of the flow, we've set the scope attribute to flow.

Start and end states

- All flows begin with a start state.
- That is to say, it exists only as a marker of where the flow should begin.
- The only thing that occurs within a start state is that a transition is performed to the next state
- In Spring Web Flow, a start state is defined in XML with a <start-state> element.
- All flows must have exactly one <start-state> to indicate where the flow should begin.

```
<start-state idref="askForPhoneNumber" />
```

- This <start-state> definition is very typical of any <start-state> in any flow definition.
- In fact, the idref attribute, which indicates the beginning state of the flow, is the only attribute available to <start-state>.
- In this case, we're indicating that the flow should begin with a state named askForPhoneNumber.
- Just as all flows must start somewhere, they all eventually must come to an end.
- Therefore, we must also define an end state in the flow:

```
<end-state id="finish" view="orderComplete" />
```

- The <end-state> element defines the hopping-off point for a flow.
- When a flow transitions to this state, there's no turning back.
- An end state terminates the flow and then displays a view specified by the view attribute.
- In this case, we've asked the flow to send the user to the view whose name is orderComplete.
- This logical view name is ultimately mapped to an actual view implementation using a Spring MVC view resolver.

The basic shell of the pizza order flow

The start and end states bookend all Spring Web Flow definitions.
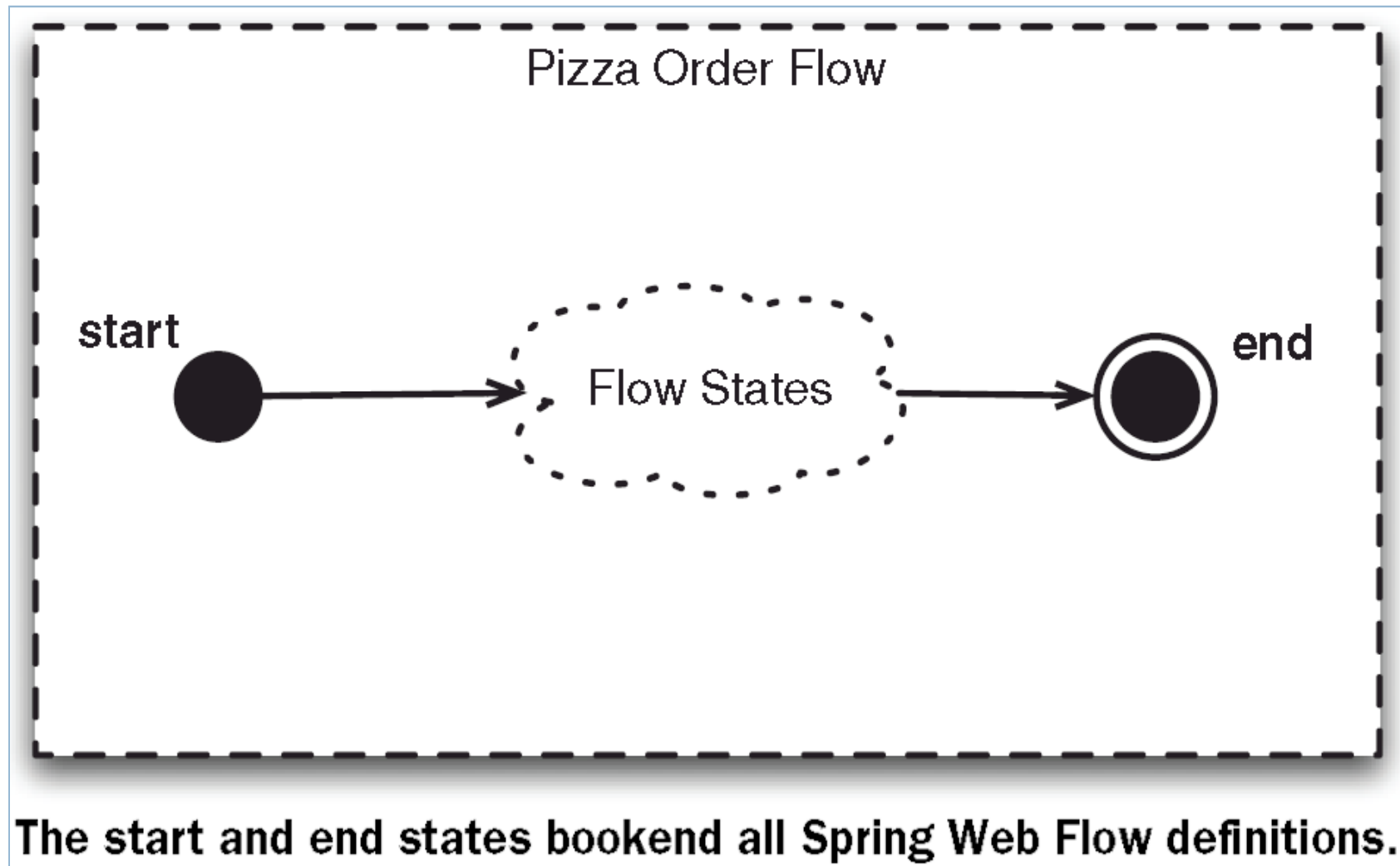
Gathering customer information
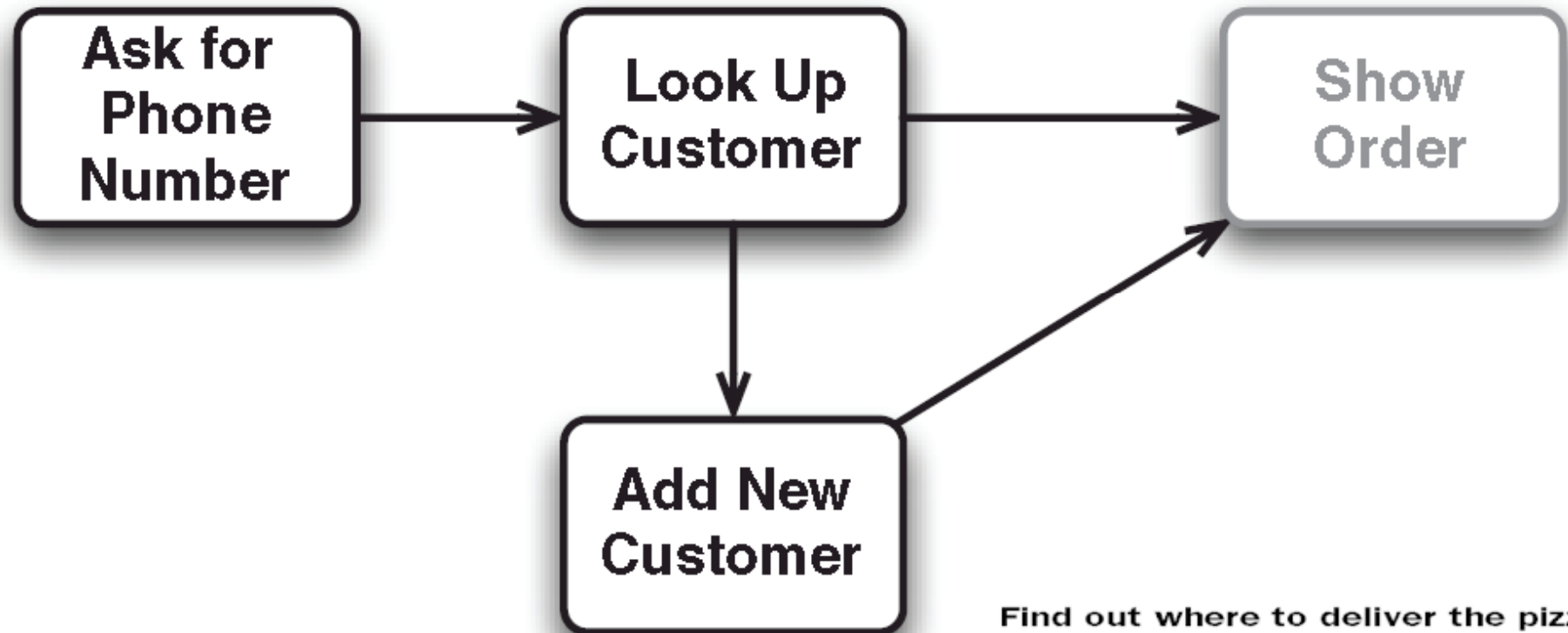
Find out where to deliver the pizza by gathering customer information in the flow.

Asking the customer for their phone number

- The first step in acquiring customer information is to ask the user for their phone number.
- This is a simple matter of presenting the user with a web page with a form for entering the phone number.
- View states are used to involve a user in a flow by displaying information and possibly asking the user for input.
- This makes a view state the perfect choice for presenting the phone number form to the user.
- The following <view-state> element should do the trick:

```
<view-state id="askForPhoneNumber" view="phoneNumberForm">
    <transition on="submit" to="lookupCustomer" />
</view-state>
```

A simplified form of phoneNumberForm.jsp

```
<h2>Customer Lookup</h2>
<form method="post" action="flow.htm">          Submits to
                                                  flow.htm
    <input type="hidden" name="_flowExecutionKey"
        value="${flowExecutionKey}">              Identifies flow

    <b>Phone number: </b>
        <input type="text" name="phoneNumber"/><br/>
    <input type="submit" class="button"
        name="_eventId_submit" value="Submit"/>    Triggers
</form>                                             submit event
```

- Recall that all links within a Spring Web Flow application should go through FlowController's URL mapping.
- Likewise, forms should also be submitted to the same URL.
- Therefore, the action parameter of the <form> element is set to flow.htm is the URL pattern previously mapped to the FlowController.
- So that FlowController will know which flow the request is for, we must also identify the flow by setting the _flowExecutionKey parameter.
- For that reason, we've added a hidden field named _flowExecutionKey that holds the flow execution key that will be submitted along with the form data.
- The final thing to note is the odd name given to the submit button.
- Clicking this button triggers an event to Spring Web Flow from a form submission.
- When the form is submitted, the name of this parameter is split into two parts.
  - The first part, _eventId, signals that we're identifying the event.
  - The second part, submit, is the name of the event to be triggered when the form is submitted.
- Looking back at the askForPhoneNumber <view-state> declaration, we see that the submit event triggers a transition to a state named lookupCustomer, where the form will be processed to look up the customer information.

Looking up customer data

- When using <view-state> to prompt the user for a phone number, we allowed the user to take part in the flow.
- Now it's the application's turn to perform some work as it attempts to look up the customer data in an action state.
- The lookupCustomer state is defined as an <action-state> with the following XML

```xml
<action-state id="lookupCustomer">
  <action bean="lookupCustomerAction" />
  <transition on="success" to="showOrder" />
  <transition on-exception="com.springinaction.pizza.service.CustomerNotFoundException"
                                                             to="addNewCustomer"/>
</action-state>
```

- The action implementation is referenced by the bean attribute of the <action> sub-element
- Here, we've specified that the functionality behind the lookupCustomer action state is defined in a bean named lookupCustomerAction.
- The lookupCustomerAction bean is configured in Spring as follows:

```xml
<bean id="lookupCustomerAction"
class="com.springinaction.pizza.flow.LookupCustomerAction">
        <property name="customerService" ref="customerService" />
</bean>
```

A flow action for looking up customer information

```java
package com.springinaction.pizza.flow;
import org.springframework.webflow.execution.Action;
import org.springframework.webflow.execution.Event;
import org.springframework.webflow.execution.RequestContext;
import com.springinaction.pizza.domain.Customer;
import com.springinaction.pizza.domain.Order;
import com.springinaction.pizza.service.CustomerService;

public class LookupCustomerAction implements Action {
  public Event execute(RequestContext context)
      throws Exception {

    String phoneNumber =                                      // Gets phone number from
        context.getRequestParameters().get("phoneNumber");    // request parameters

    Customer customer =                                       // Looks up
        customerService.lookupCustomer(phoneNumber);          // customer

    Order order =
        (Order) context.getFlowScope().get("order");          // Sets customer to
    order.setCustomer(customer);                              // flow-scoped order

    return new Event(this, "success");                        // Returns
  }                                                           // success event

// injected
private CustomerService customerService;
public void setCustomerService(
    CustomerService customerService) {
  this.customerService = customerService;
  }
}
```

- LookupCustomerAction assumes that it will be processing the submission of the askForPhoneNumber view state and retrieves the phone number from the request parameters.
- It then passes that phone number to the lookupCustomer() method of the injected CustomerService object to retrieve a Customer object.
- If a Customer is found, the pizza order is retrieved from flow scope and its customer property is set accordingly.
- At this point, we have all of the customer information we need, so we're ready to start adding pizzas to the order.
- So the execute() method concludes by returning a success event.
- Looking back at the definition of the lookupCustomer <action-state>, we see that a success event results in a transition to the showOrder event.
- If, however, lookupCustomer() can't find a Customer based on the given phone number, it will throw a CustomerNotFoundException.
- In that case, we need the user to add a new customer.
- Therefore, the lookupCustomer <action-state> is also declared with an exception transition that sends the flow to the addNewCustomer state if a CustomerNotFoundException is thrown from LookupCustomerAction's execute() method.

Adding a new customer

```
<view-state id="addNewCustomer" view="newCustomerForm">
…
</view-state>
```

As configured here, the addNewCustomer view state will display the view defined in /WEB-INF/jsp/newCustomerForm.jsp

A form for creating a new customer

```
<%@ taglib prefix="form"
    uri="http://www.springframework.org/tags/form" %>

<h2>New customer</h2>
<form:form action="flow.htm"                          Submits form to
    commandName="order.customer">                     FlowController
  <input type="hidden" name="_flowExecutionKey"
      value="${flowExecutionKey}"/>                   Identifies current
                                                      flow execution

  <b>Phone:  </b> ${requestParameters.phoneNumber} <br/>
  <b>Name:  </b> <form:input path="name" /><br/>
  <b>Street address:  </b>
      <form:input path="streetAddress" /><br/>       Uses Spring
  <b>City:  </b> <form:input path="city" /><br/>      form-binding
  <b>State:  </b> <form:input path="state" /><br/>    JSP tags
  <b>Zip:  </b> <form:input path="zipCode" /><br/>
  <input type="submit" class="button"                 Submits form and
      name="_eventId_submit" value="Submit"/>         fires submit event
  <input type="submit" class="button"
      name="_eventId_cancel" value="Cancel"/>         Submits form and
</form:form>                                           fires cancel event
```

- Although the addNewCustomer state will display the new customer form, it's not able to process the form data.
- Eventually the user will submit the form and we'll need a way to bind the form data to a back-end Customer object.
- Fortunately, Spring Web Flow provides FlowAction, a special Spring Web Flow Action implementation that knows how to deal with common form-binding logic.
- To use FlowAction, the first thing we'll need to do is configure it as a <bean> in the Spring application context:

```
<bean id="customerFormAction" class="org.springframework.webflow.action.FormAction">
  <property name="formObjectName" value="customer" />
  <property name="formObjectScope" value="REQUEST" />
  <property name="formObjectClass" value="com.springinaction.pizza.domain.Customer" />
</bean>
```

FormAction

- FormAction has three important properties that describe the object that will be bound to the form.
- The formObjectName property specifies the name of the object, formObjectScope specifies the scope, and formObjectClass specifies the type.
- In this case we're asking FormAction to work with a Customer object in request scope as customer.
- When the form is first displayed, we'll need FormAction to produce a blank Customer object so that we'll have an object to bind the form data to.
- To make that happen, we'll add FormAction's setupForm() method as a render action in the addNewCustomer state:

```
<view-state id="addNewCustomer" view="newCustomerForm">
<render-actions>
<action bean="customerFormAction" method="setupForm"/>
</render-actions>

…
</view-state>
```

- Render actions are a way of associating an action with the rendering of a view.
- In this case, the FormAction's setupForm() method will be called just before the customer form is displayed and will place a fresh Customer object in request scope.
- In a sense, FormAction's setupForm() is a lot like a Spring MVC form controller's formBackingObject() in that it prepares an object to be bound to a form.

Transition

- When the new customer form is submitted, we'll need a transition to handle the submit event.
- The following <transition> addition to addNewCustomer describes what should happen:

```
<view-state id="addNewCustomer" view="newCustomerForm">
    <render-actions>
        <action bean="customerFormAction" method="setupForm"/>
    </render-actions>
    <transition on="submit" to="showOrder">
            <action bean="customerFormAction" method="bind" />
            <evaluate-action expression="flowScope.order.setCustomer(requestScope.customer)" />
    </transition>
</view-state>
```

- The <transition> element itself is mostly straightforward. It simply says that a submit event should trigger a transition to the state named showOrder.
- But before we go to the showOrder state, we must first bind the form data to the form-backing Customer object and set the customer property of the flow-scoped Order. This is a two-step process:
    - First, we use an <action> element to ask FormAction's bind() method to bind the submitted form data to the form-backing Customer object that is in request scope.
    - Next, with a fully populated Customer object in request scope, we use <evaluate-action> to copy the request-scoped Customer object to the Order object's customer property.

Pizza order flow now has all of the customer information gathering states it needs
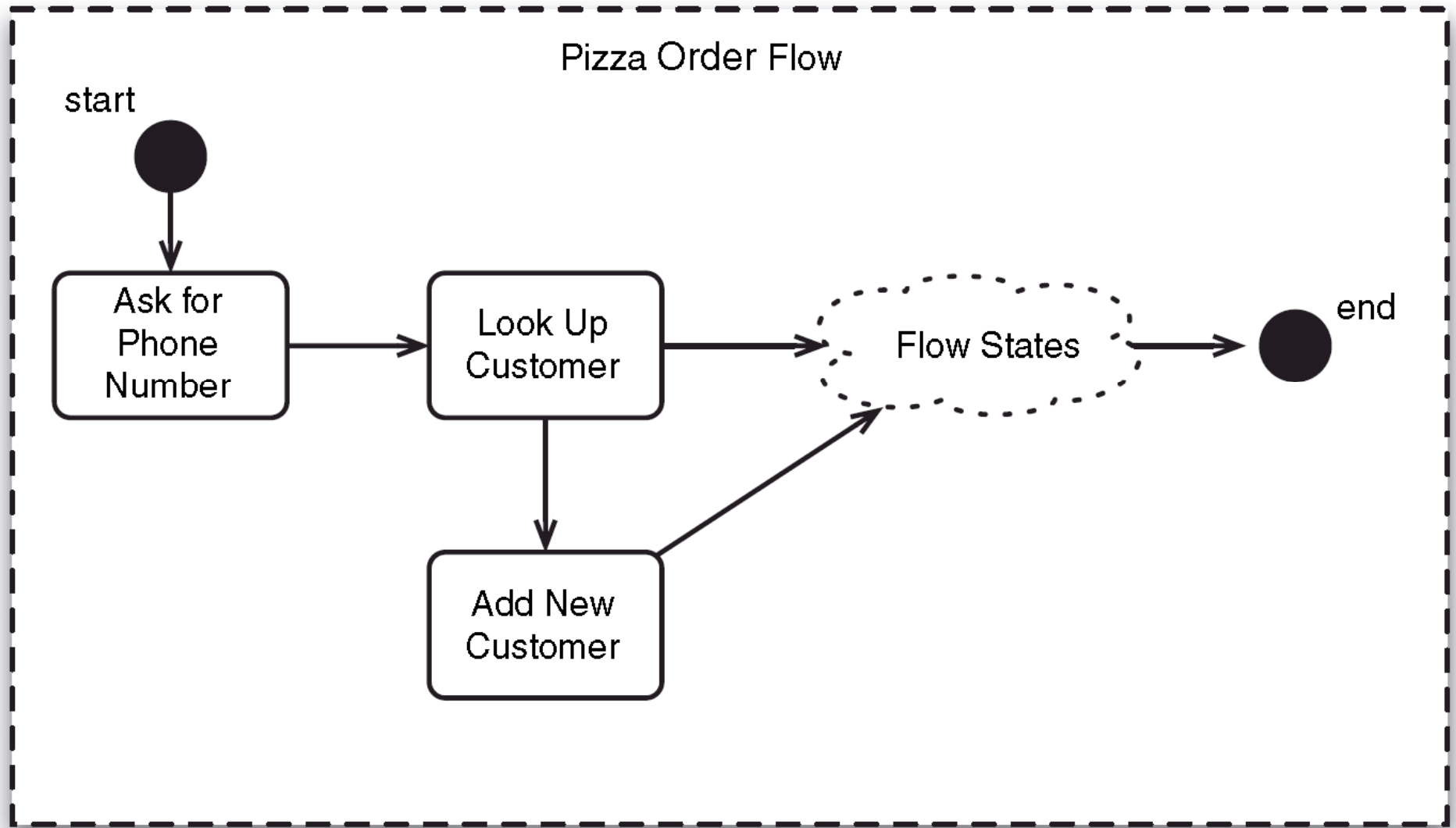
Building a pizza order

- When it comes to defining a flow for creating a pizza order, the most critical part is where the pizzas get added to the order.
- Without that, we're not really delivering anything to the customer.
- So, the next couple of states we introduce to the flow will serve to build the pizza order.
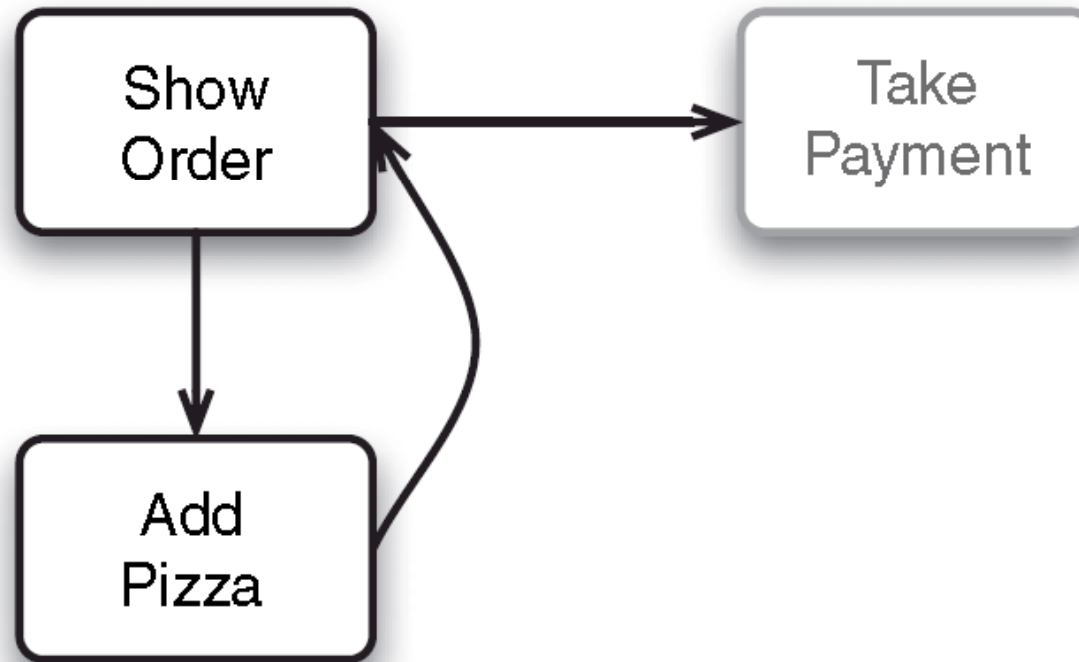
Adding states to the flow to build the pizza order

Displaying the order

```
<view-state id="showOrder" view="orderDetails">
  <transition on="addPizza" to="addPizza" />
  <transition on="continue" to="takePayment" />
</view-state>
```

There's nothing particularly special about this `<view-state>` definition. Compared to some `<view-state>`s we've seen already, this one is plain vanilla. It simply renders the view whose name is `orderDetails`. `InternalResourceViewResolver` will resolve this view name to /WEB-INF/jsp/orderDetails.jsp, which is the JSP file

orderDetails.jsp, which displays the current pizza order to the user

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="fmt"
    uri="http://java.sun.com/jstl/fmt" %>

<h2>${order.customer.name}</h2>                        Displays
<b>${order.customer.streetAddress}</b><br/>            customer
<b>${order.customer.city}, ${order.customer.state}     info
    ${order.customer.zipCode}</b><br/>
<b>${order.customer.phoneNumber}</b><br/>
<br/>

<a
   href="flow.htm?_flowExecutionKey=${flowExecutionKey}   Links to fire
     ➥ &_eventId=continue">Place Order</a>               continue event

<a href="flow.htm?_flowExecutionKey=${flowExecutionKey}  Links to fire
     ➥ &_eventId=cancel">Cancel</a>                      cancel event
<hr/>
<h3>Order total: <fmt:formatNumber type="currency"
    value="${order.total}"/></h3>
  <hr/>
  <h3>Pizzas:</h3>
  <small>
    <a href="flow.htm?_flowExecutionKey=${flowExecutionKey}  Links to fire
       ➥ &_eventId=addPizza">Add Pizza</a>               addPizza event
  </small>
  <br/>
  <c:forEach items="${order.pizzas}" var="pizza">
  <li>${pizza.size} :                                    Displays
    <c:forEach items="${pizza.toppings}" var="topping">  pizzas
        ${topping},
    </c:forEach>
  </li>
  </c:forEach>
```

☐ Notice that all three of these links are very similar. Consider the addPizza link, for example:

AddPizza

☐ There are three very important elements of the link's href attribute that guide Spring Web Flow:

■ As we've discussed before, all links within a flow must go through the Flow-Controller. Therefore the root of the link is flow.htm, the URL pattern mapped to the FlowController.

■ To identify the flow execution to Spring Web Flow, we've set the _flowExecutionKey parameter to the page-scoped ${flowExecutionKey} variable. This way FlowController will be able to distinguish one user's flow execution from another.

■ Finally, the _eventId parameter identifies the event to fire when this link is clicked on. In this case, we're firing the addPizza event, which, as defined in the showOrder state, should trigger a transition to the addPizza state.

Adding a pizza to the order

- Since we'll be prompting the user to choose a pizza, it makes sense for the addPizza state to be a view state. Here's the <view-state> definition we'll use:

```xml
<view-state id="addPizza" view="newPizzaForm">
        <render-actions>
                <action bean="pizzaFormAction" method="setupForm"/>
        </render-actions>
        <transition on="submit" to="showOrder">
                <action bean="pizzaFormAction" method="bind" />
                <evaluate-action expression="flowScope.order.addPizza(requestScope.pizza)" />
        </transition>
</view-state>
```

- If this <view-state> definition looks familiar, it's because we used a similar pattern when adding a new customer.
- Just as with the customer form, we're using a FormAction to set up and bind the form data.
- As you can see from the definition of the pizzaFormAction bean, this time the form-backing object is a Pizza object:

```xml
<bean id="pizzaFormAction" class="org.springframework.webflow.action.FormAction">
        <property name="formObjectName" value="pizza" />
        <property name="formObjectClass" value="com.springinaction.pizza.domain.Pizza" />
        <property name="formObjectScope" value="REQUEST" />
</bean>
```

When the new pizza form is submitted, the `<evaluate-action>` copies the request-scoped `Pizza` into the order by calling the flow-scoped `Order` object's
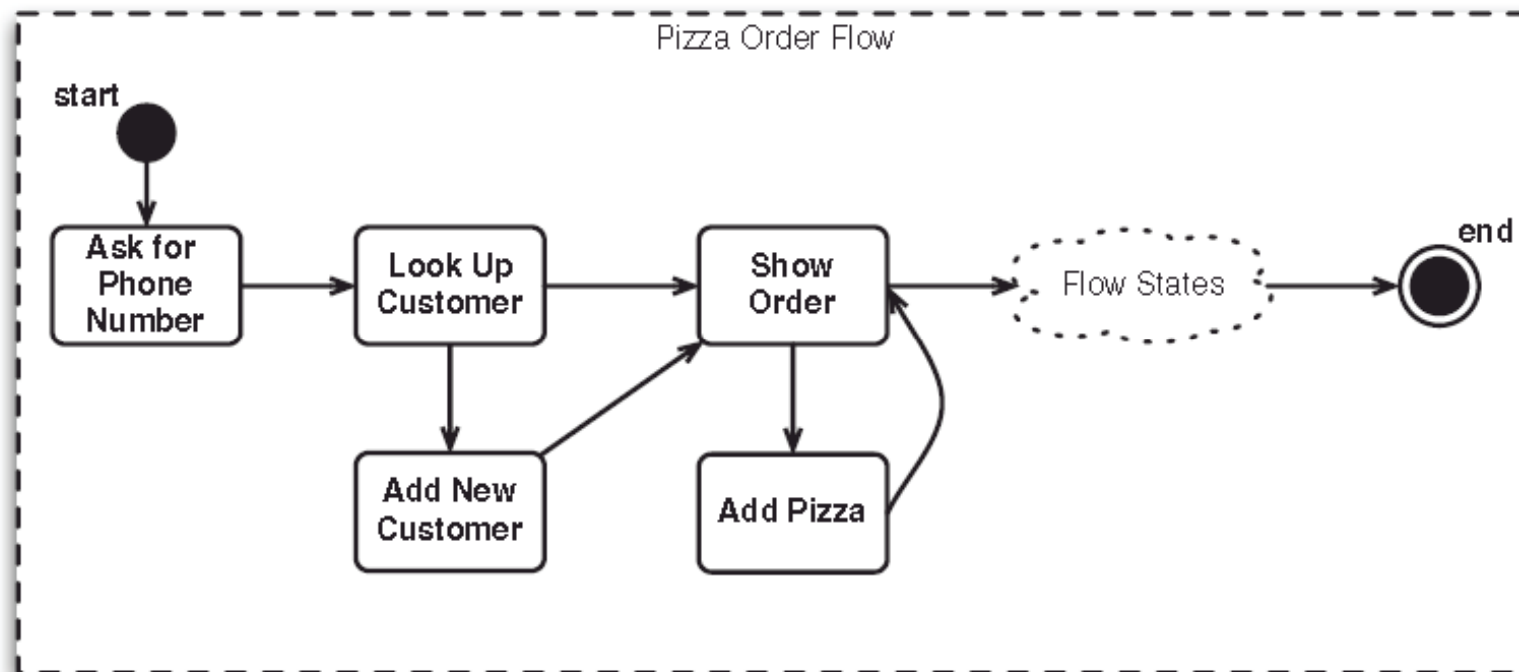

addPizza() method. Once the pizza has been added to the order, the flow transitions back to the showOrder view state to display the order's status.

Taking payment

In order to take payment, we must prompt the user for credit card information. Since we're asking the user to get involved again, this means that a view state is in order. The following `<view-state>` displays the payment form to the user and processes the credit card payment:

```xml
<view-state id="takePayment" view="paymentForm">
  <transition on="submit" to="submitOrder">
    <bean-action bean="paymentProcessor"
        method="approveCreditCard">
      <method-arguments>
        <argument expression=
            "${requestParameters.creditCardNumber}"/>
        <argument expression=
            "${requestParameters.expirationMonth}"/>
        <argument expression=
            "${requestParameters.expirationYear}"/>
        <argument expression=
            "${flowScope.order.total}" />
      </method-arguments>
    </bean-action>
  </transition>
  <transition on-exception=
        "com.springinaction.pizza.PaymentException"
      to="takePayment" />
</view-state>
```
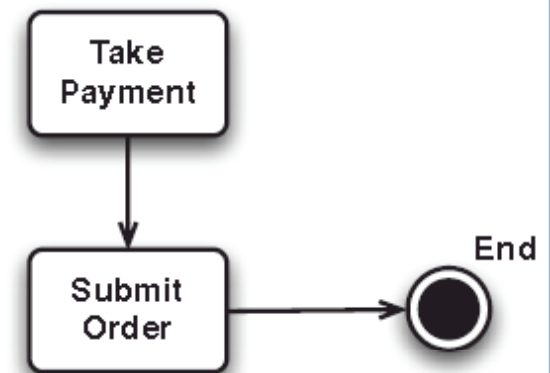

The final two states in the flow take payment and submit the order.

Submitting the order

Finally, we're ready to submit the order. The user doesn't need to be involved for this part of the flow, so an `<action-state>` is a suitable choice. The following `<action-state>` saves the order and finishes the flow:

```
<action-state id="submitOrder">
  <bean-action bean="orderService" method="saveOrder">
    <method-arguments>
      <argument expression="${flowScope.order}" />
    </method-arguments>
  </bean-action>

  <transition on="success" to="finish" />
</action-state>
```

Notice that we're using a `<bean-action>` to define the logic behind the `<action-state>`. Here, the `saveOrder()` method will be called on the bean whose name is `orderService`. The flow-scoped `Order` object will be passed in as a parameter. This means that the class behind the `orderService` bean must expose a `saveOrder()` method whose signature looks like this:

```
public void saveOrder(Order order) {
…
}
```

As before, the actual implementation of this method isn't relevant to the discussion of Spring Web Flow, so we've purposefully omitted it to avoid confusion.

The flow now appears to be complete. We have all of the states in place and we should be able to start taking pizza orders. But we're missing one small transition that is used throughout the entire flow.

A few finishing touches

On more than one occasion, we've seen links that fire `cancel` events within the flow, but we've never told you how those `cancel` events are handled. Up to now, we've been avoiding the issue, but now it's time to meet it head on.
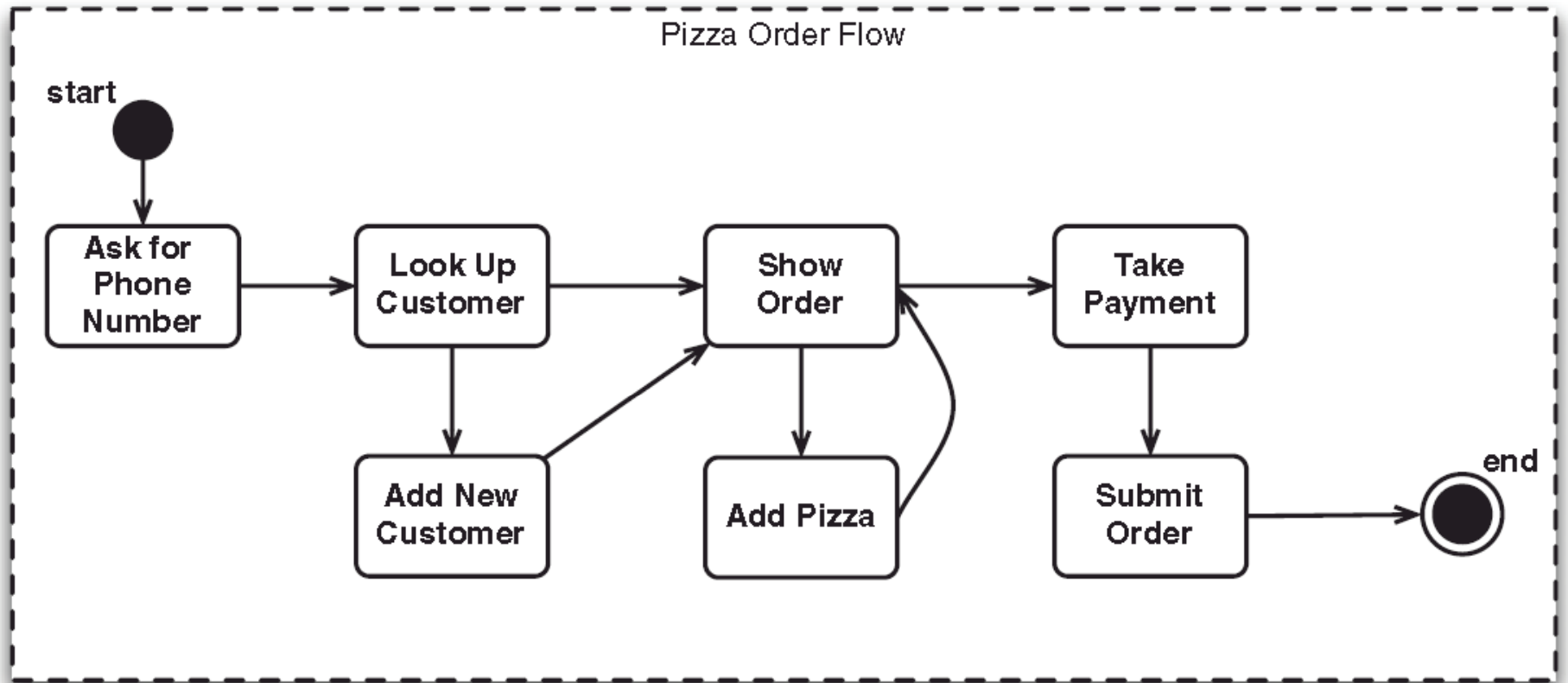
At any view state within the flow, the customer may choose to cancel the order and start over. When that happens, we need the flow to transition to the `finish` state to close down the flow execution. A naive way of handling `cancel` events is to place appropriate `<transition>`s all throughout the flow. For example, consider the `cancel` transition that *could* have been added to the `showOrder` state:

```
<view-state id="showOrder" view="orderDetails">
  <transition on="addPizza" to="addPizza" />
  <transition on="continue" to="takePayment" />
  <transition on="cancel" to="finish" />
</view-state>
```

The problem with doing it this way is that we must copy the exact same `<transition>` element to all of our flow's `<view-state>`s. Our flow is simple enough that duplicating the `<transition>` wouldn't be too painful. Nonetheless, it still results in an undesired duplication of code and should be avoided.

Instead of defining the same `<transition>` multiple times throughout an entire flow, Spring Web Flow offers the ability to define global transitions. Global transitions are transition definitions that are applicable from any flow state. To add global transitions to a flow, simply add a `<global-transitions>` element as a child of the `<flow>` element and place `<transition>` elements within it. For example:

```
<global-transitions>
  <transition on="cancel" to="finish" />
</global-transitions>
```

Pizza Order Flow

start

Ask for Phone Number → Look Up Customer → Show Order → Take Payment

Look Up Customer → Add New Customer → Show Order

Show Order → Add Pizza → Show Order

Take Payment → Submit Order → end

The pizza order flow is now complete

Advanced web flow techniques

- Although the pizza order flow is now complete and ready to take orders for the delicious Italian pies, there is still some room for improvement.

- Specifically, there are two improvements in mind:

  - Once we have a customer's information on hand, we should determine whether or not they live within our delivery area.

  - The customer information portion of the flow may come in handy for other flows. So, it would be nice to extract that portion of the flow into a subflow that can be reused in another flow (perhaps for a flower delivery service).

- Coincidentally, these improvements involve the two flow states that we haven't talked about yet: decision states and subflow states.

- That makes this an opportune time to give those two flow states a try.

Using decision states

After the `lookupCustomer` and `addNewCustomer` states and just before the `showOrder` state, we have a decision to make: can we deliver the pizza to the customer's given address?

More accurately, our flow has a decision to make. If the customer lives within the delivery area then there's no problem. The flow should proceed normally. But if the customer lives outside of the delivery area, we should transition to a warning page to indicate that we can't deliver pizza to the customer's address but that they are welcome to place the order for carryout and pick it up themselves.

When flow-diverging decisions must be made, a decision state is in order. A decision state is the Spring Web Flow equivalent of an `if/else` statement in Java. It evaluates some Boolean expression and based on the results will send the flow in one of two directions.

Decision states are defined with the `<decision-state>` element. The following `<decision-state>` is what we'll use to decide whether or not to warn the user that they are out of the delivery area:

```
<decision-state id="checkDeliveryArea">
  <if test="${flowScope.order.customer.inDeliveryArea}"
      then="showOrder"
      else="warnNoDeliveryAvailable"/>
</decision-state>
```

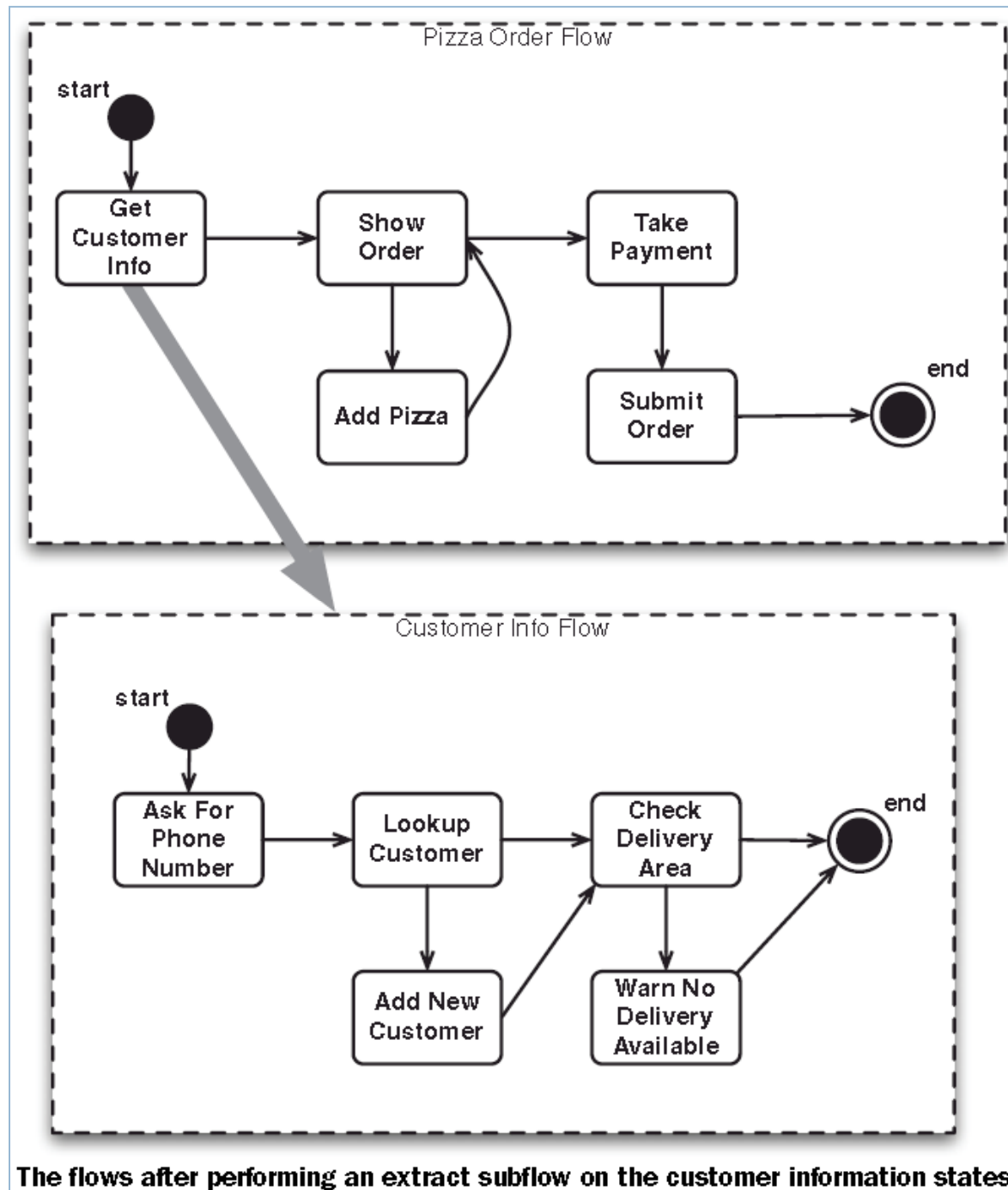# The new flow diagram after adding the delivery area check
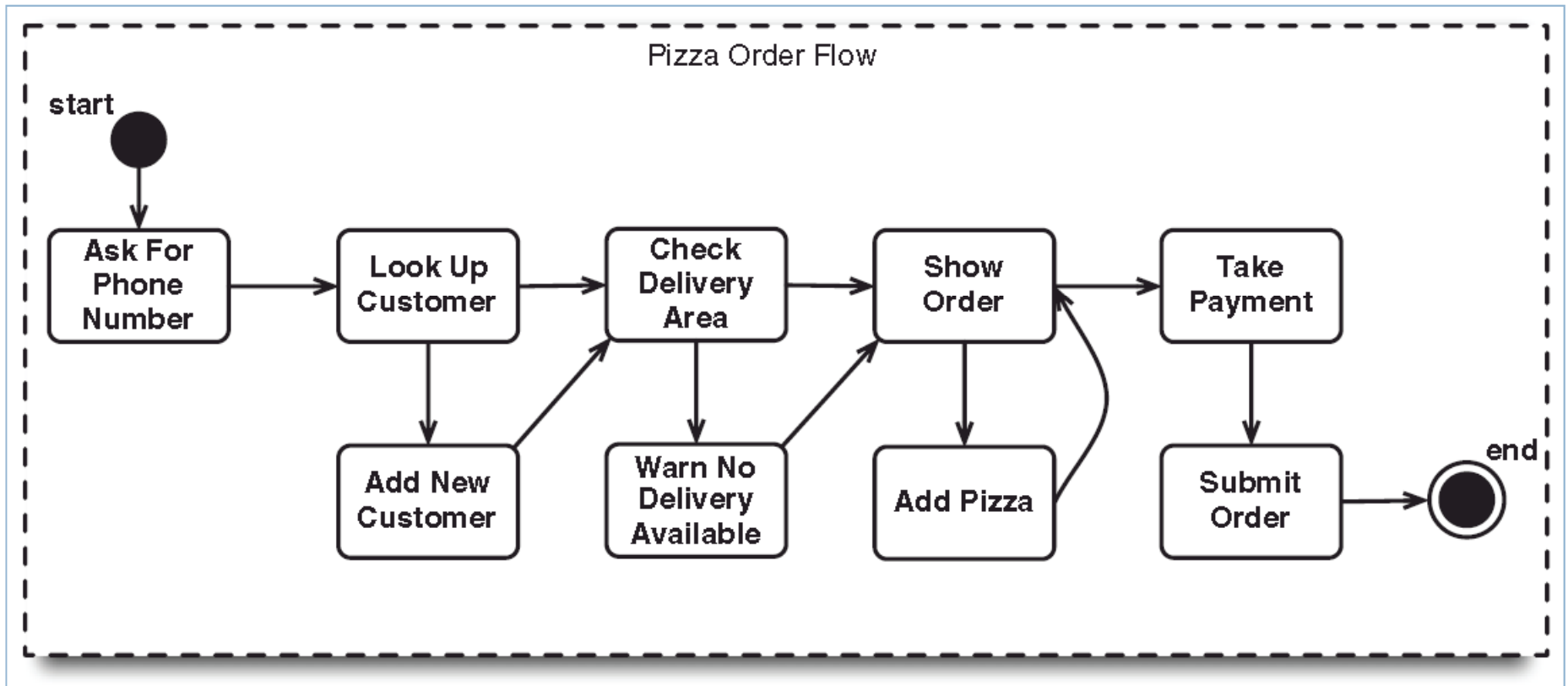
Extracting subflows and using substates

You've probably noticed by now that the flow definition file is getting a bit lengthy. Counting the two flow states that we added in the previous section and the start and end states, our pizza order flow's state count is up to 11. Although this is far from being the most complex flow ever defined, it is starting to get unwieldy.

In Java, when a method gets too big, it is often beneficial to pull out related lines of code and place them into their own method. In his *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999), Martin Fowler refers to this technique as an "extract method." By performing an extract method refactoring, lengthy methods become shorter and easier to follow.

Fowler's book doesn't cover flow definitions per se. Nevertheless, the idea behind extract methods applies equally well to flow definitions. Flow definitions can get quite lengthy and difficult to follow. It may be advantageous to extract a related set of states into their own flow and to reference that flow from the main flow. As with methods, the reward is smaller and easier-to-understand flow definitions.

Pizza Order Flow

start

Get Customer Info → Show Order → Take Payment

Show Order → Add Pizza → Show Order

Take Payment → Submit Order → end

Customer Info Flow

start

Ask For Phone Number → Lookup Customer → Check Delivery Area → end

Lookup Customer → Add New Customer → Check Delivery Area

Check Delivery Area → Warn No Delivery Available → end

The flows after performing an extract subflow on the customer information states.

Example: The Login Flow

- This example is fairly simple and works along with a flow for registering the user.
- This flow is invoked upon an attempt to log into the PixWeb sample application.
- The login flow lives in the WEB-INF/flows directory in a file named login-flow.xml.
  (Notice that the location and filename both match the pattern from the SWF configuration shown earlier.)
- It does not stand alone, as it has a companion file that will be discussed briefly in this section.

  ```
  <flow>
  <start-actions>
  <action bean="loginAction" method="setupForm"/>
  </start-actions>

  . . . . .
  ```

- The login flow begins with a start-actions element that contains a definition for an action bean named loginAction and points to a method named setupForm.
- First, the loginAction bean is defined in another file named login-flow-beans.xml that is imported at the bottom of the login-flow.xml file; you will look at this in a moment.
- For the time being, know that a flow can be read using natural language.
- For example, the flow here says, "Upon initialization of this flow, execute the setupForm() method on the loginAction bean."
- The setupForm method is part of the SWF FormAction class, which creates any form objects and their validators and errors objects as well as any necessary property editors.
- All of these objects are stored in various scopes accessible from the request context.

login-flow.xml

```xml
<flow>
<start-actions>
    <action bean="loginAction" method="setupForm"/>
</start-actions>
<start-state idref="login" />
<view-state id="login" view="login">
    <transition on="submit" to="userLookup" />
</view-state>
<action-state id="userLookup">
    <action bean="loginAction" method="login" />
    <transition on="success" to="finish" />
    <transition on="error" to="registrationFlow" />
</action-state>
    . . . . . .
```

- All flows define a start-state: this one points to a view state named login, which points to a view named login and defines a transition.
- This portion of the login flow says, "Display a view named login, and when it submits an event named submit, advance control to the action state named userLookup.
- When userLookup is invoked, call the login method on a bean named loginAction.
- If a success event occurs, advance control to a state named finish; if an error event occurs, advance control to a state named registrationFlow."

```xml
<flow>
<start-actions>
    <action bean="loginAction" method="setupForm"/>
</start-actions>
<start-state idref="login" />
<view-state id="login" view="login">
    <transition on="submit" to="userLookup" />
</view-state>
<action-state id="userLookup">
    <action bean="loginAction" method="login" />
    <transition on="success" to="finish" />
    <transition on="error" to="registrationFlow" />
</action-state>
<subflow-state id="registrationFlow" flow="registration-flow">
    <attribute-mapper>
        <input-mapper><input-attribute name="user"/></input-mapper>
        <output-mapper><output-attribute name="user" /></output-mapper>
    </attribute-mapper>
    <transition on="finish" to="login" />
</subflow-state>
<end-state id="finish" view="flowRedirect:albums-flow" />
<import resource="login-flow-beans.xml" />
</flow>
```

login-flow.xml

- The registrationFlow is a subflow that passes control to a completely different flow named registration-flow, which will be discussed in the next slides.
- So what is added here is simply a hook to call this flow that has not yet been created.
- This section says, "Launch another flow in the flow registry named registration-flow, pass in the user object and map the ouptut of the subflow to the user object.
- When the registration-flow triggers an event named finish, pass control back to the state named login."
- The registration-flow subflow is defined in another file that we'll review in a moment.
- Not only do subflows promote reuse but reuse encourages a better overall design.
- A Java API is usually better designed if the designer is thinking in terms of reuse (instead of solitary use), and the design of flows follows the same paradigm.
- Flows that are designed for reuse are not monolithic but are fairly targeted at solving a particular problem and this typically causes them to remain smaller and therefore easier to maintain.
- So it's a good idea to make use of subflows wherever possible.
- There is also an end state named finish.
- This state defines a view but instead of simply pointing to another view (such as another JSP), it uses a trick to do a flow redirect to the albums-flow, another flow defined in another file named albums-flow.xml.
- The flow redirect causes a brand-new execution of the flow that is named and passes control to it.
- A subflow, on the other hand, spawns a new flow but always returns control to the parent flow.
- This is a nice little trick when you need to pass control to a different flow but you don't want to use a subflow.
- In addition, the very last element in the login-flow is a standard Spring beans import element that imports a file named login-flow-beans.xml. This is the companion file mentioned earlier in this section.

The Login Flow Beans XML

- This companion file is named login-flow-bean.xml and is used to define bean definitions specific to the login flow.

```xml
<beans>
    <bean name="validator" class="com.yusuf.pix.validation.PixUserValidator" />
    <bean name="loginAction" class="com.yusuf.pix.action.LoginAction"
                                    p:validator-ref="validator" p:user-repo ref="userRepo"/>
</beans>
```

- The login flow beans file is just another Spring beans configuration file that instantiates the LoginAction bean and a dependency bean.

- It's a separate file simply to demonstrate a best practice to enforce a separation of concerns.

- Quite simply, the beans defined in this file are only needed from the login flow, so they're defined in this separate configuration.